(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 645 699 A1**

(12) **EUROPEAN PATENT APPLICATION**

(21) Application number: **93115706.9**

(51) Int. Cl.6: **G06F 7/544**

(22) Date of filing: **29.09.93**

(43) Date of publication of application:
29.03.95 Bulletin 95/13

(84) Designated Contracting States:
DE FR GB

(71) Applicant: **International Business Machines Corporation**
Old Orchard Road
Armonk, N.Y. 10504 (US)

(72) Inventor: **Dao-Trong, Son, Dr.**
Schönbergstrasse 2
D-70599 Stuttgart (DE)
Inventor: **Haas, Jürgen**
Daimlerstrasse 6
D-72074 Tübingen (DE)
Inventor: **Müller, Rolf**
Hauffstrasse 8/1
D-71032 Böblingen (DE)

(74) Representative: **Jost, Ottokarl, Dipl.-Ing.**
IBM Deutschland Informationssysteme GmbH,
Patentwesen und Urheberrecht
D-70548 Stuttgart (DE)

(54) Fast multiply-add instruction sequence in a pipeline floating-point processor.

(57) The present invention is related to a pipeline floating point processor in which the addition pipelining is reorganized so that no wait cycle is needed when the addition uses the result of an immediately foregoing multiplication (fast multiply-add instruction).

The re-organization implies the following changes of an existing data flow of the pipeline floating processor shown in Fig. 4:

1. Data feed-back via path ND of normalized data from the multiplier M into the aligners AL1,2;
2. Shift left one digit feature on both sides of the data path for taking account of a possible leading zero digit of the product, and special zeroing of potential guard digits by Z1,2;
3. Exponent build by 9 bits for overflow and underflow recognition, and due to an underflow the exponent result is reset to zero on the fly by a true zero unit (T/C).

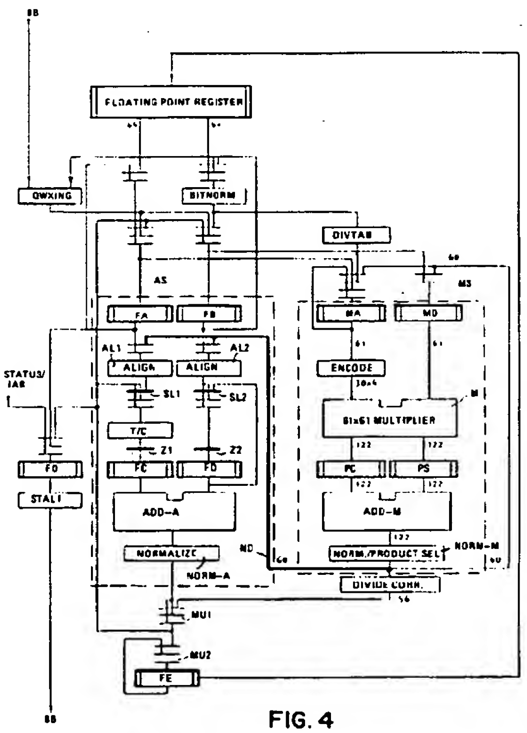


FIG. 4

The invention is related to an arrangement and a method in a pipeline floating-point processor (FLPT) of improving the performance of a multiply-add sequence in which the multiplication is performed within three cycles: operand read, partial sums build, and add the partial sums to end result, and where the addition also needs three cycles: operand read, operands alignment, and addition.

Floating-point processors (FLPTs) are used to be functionally added to a main processor (CPU) for performing scientific applications. In the entry-level models (e.g. 9221) of the IBM Enterprise System/9000 (ES/9000) the floating-point processor is tightly coupled to the CPU and carries out all IBM System/390 floating-point instructions. All instructions are hardware-coded, so no microinstructions are needed. Moreover, binary integer multiplication is also implemented on the floating-point unit to improve overall performance.

Fig.1 shows the data flow of the above mentioned floating point processor which is described in more detail in the IBM Journal of Research and Development, Vol. 36, Number 4, July 1992. While the CPU is based on a four stage pipeline, the floating-point processor requires a five stage pipeline to perform its most used instructions, e.g. add, subtract, and multiply in one cycle for double-precision operands (reference should be made to "ESA/390 Architecture", IBM Form No.: G580-1017-00 for more detail).

The CPU resolves operand addresses, provides operands from the cache, and handles all exceptions for the floating-point processor. The five stages of the pipeline are instruction fetch, which is executed on the CPU, register fetch, operand realignment, addition, and normalization and register store.

To preserve synchronization with the CPU, a floating-point wait signal is raised whenever a floating-point instruction needs more than one cycle. The CPU then waits until this wait signal disappears before it increments its program counter and starts the next sequential instruction, which is kept on the bus.

Because the IBM System/390 architecture requires that interrupts be precise, a wait condition is also invoked whenever an exception may occur. As can further be seen from Fig.1 many bypass busses are used to avoid wait cycles when the results of the foregoing instructions are used. A wait cycle is needed only if the result of one instruction is used immediately by the next sequential instruction (NSI), e.g. when an add instruction follows a multiply instruction, the result of which has to be augmented by the addend of the add instruction.

The data flow shown in Fig.1 has two parallel paths for fraction processing: one add-path where all non-multiply/divide instructions are implemented, and one multiply path specially designed for multiply and divide. The add-path has a fixed (60) bit width and consists of an operand switcher, an aligner, an adder, and a normalizer shifter. Instead of using two aligners on each side of the operand paths, a switcher is used to switch operands, thereby saving one aligner. The switcher is also needed for other instructions, and so, requires much fewer circuitry.

The multiplier path consists of a booth encoder for the 58-bit multiplier, a multiplier macro which forms the 58x60-bit product terms sum and carry, and a 92-bit adder which delivers the result product. The sign and exponent paths are adjusted to be consistent with the add path. The exponent path resolves all exception and true zero situations, as defined by the earlier cited IBM System/390 architecture.

The implementation of all other instructions is merged into the add path and multiply path, and requires only minimal additional logic circuits. The data flow in Fig.1 therefore, shows more function blocks and multiplexer stages than needed for only add, subtract, and multiply operations.

As further can be seen from Fig.1, the data flow is partitioned into smaller parts FA, FB, FC, FD, MA, MB, PS, PC, and PL (typically registers with their input control). These partitions and the partitioning of the floating-point instructions into three main groups are:

1.) addition/subtraction, load;
2.) multiplication; and
3.) division.

These are the instructions most used in scientific applications. The first two groups of instructions are performed in one cycle, and division is made as fast as possible.

For an add instruction, during the first two pipeline stages, only instruction and operand fetching are done. All data processing is concentrated in the third and fourth pipeline stages. In the fifth stage, the result is written back to a floating-point register.

Loading operations are treated like addition, with one operand equal to zero. During stage 3 the exponents of both operands are compared in order to determine the amount of alignment shift. The operand with the smaller exponent is then passed to the aligner for realignment. In stage 4 of the pipeline the aligned operands are added. The addition may produce a carry-out, which results in a shift right by one digit position, in accordance with the said architecture. The exponent is then decreased accordingly.

Since time is still available in stage 4, the exponent calculation is made sequentially after that of addition, using only one exponent adder with an input multiplexer (Fig.1) to select whether an exponent increase, an exponent adjustment, or a multiply/divide exponent is required.

Leading-zero detection is made by calculating the hexadecimal digit sums without a propagated carry-in. Hexadecimal sums 0 and F for the digit position i are determined and fed into a multiplexer. The carry-in to this digit position selects whether or not the result digit is zero. This carry bit comes from the same carry-look ahead circuit used for the adder, so no additional circuit is needed. By using the above described logic, the shift amount can be determined at nearly the same time as the addition result.

Exponent exception, either overflow or underflow, is also detected in stage 4. Meanwhile, the next instruction has already been started. As earlier mentioned, a wait may be raised at stage 3 to hold execution of the next sequential instruction. In the case of an effective addition, the wait situation is met when

- the intermediate result exponent is 7F (hex) and will overflow when an exponent increment is caused by a carry-out from the adder;
- the intermediate result exponent is smaller than 0D, and a normalization is required for unnormalized operands.

Here the exponent must be decreased by the normalization shift amount, which can be at most 0D (decimal 14), thus producing an exponent underflow.

Multiplication is implemented by using a modified Booth algorithm multiplier with serial addition of partial product terms. It is used to be performed within three instruction cycles in most of the high performance mathematical co-processors:

1. Operand read,
2. Partial sums build and
3. Add partial sums to the end result. (Reference should be made to Fig.2b)

Data bypass in the first and third cycles allows a saving of one cycle when using the same result for a following instruction. However, one wait cycle is still needed as can be seen from Fig.3, where a multiply instruction is immediately followed by an add instruction which uses as addend or augment the result of the preceding multiplication.

In solving mathematical problems, especially in matrix calculations, the sequence multiply-add, where the add operation uses the result of the multiplication, is used very often.

Risc (reduced instruction set computer) systems, such as IBMs RS 6000, have a basical design which allows the combination of both operations in a single complex. However, this design is not conform with the ESA/390 architecture earlier cited. Old programs may deliver different results as from ESA/390 mode. To avoid this a single wait cycle has to be inserted (Fig.3).

In performance calculations the LINPACK loop is used very often, which consists of a sequence of five instructions:

1.) Load;
2.) Multiply;
3.) Add;
4.) Store; and
5.) Branch back.

The branch instruction is normally processed in zero-cycle so that the additional wait cycle would contribute to a performance degradation of 25%.

So, it is the object of this invention to increase the performance of pipeline floating-point processors, mainly when matrix calculations have to be performed, with their high quantity of multiply-add sequences using the result of the immediately preceding multiplication.

This object of the invention is accomplished for an arrangement by the features of claim 1 and for a method by the features of claim 2.

By applying the above features on a pipeline floating-point processor the advantage of a 25% performance increase for multiply-add instructions will be achieved.

A full understanding of the invention will be obtained from the detailed description of the preferred embodiment presented herein below, and the accompanying drawings, which are given by way of example, wherein

Fig.1	illustrates a block diagram of a prior art pipeline floating-point processor;
Figs.2a, 2b, 3	show a schematic representation of various stages of a pipeline handling an add instruction, a multiply instruction, and a multiply-add instruction sequence in a pipeline floating-point processor of Fig. 1;
Fig.4	illustrates a block diagram of a pipeline floating-point processor, modified in accordance with the invention; and
Fig.5	shows a schematic representation of the pipeline stages of a floating-point processor of Fig.4, handling a multiply-add instruction sequence;
Fig.6 - 11	depict various examples of conventional add and multiply operations in a pipeline floating processor of Fig.4; and
Fig.12 - 15	depict various examples of the new multiply-add instruction in a pipeline floating point processor of Fig.4.

The new data flow of a pipeline floating-point processor shown in Fig.4 allows a zero wait processing of the multiply-add instruction sequence, as can be seen from Fig.5, which is obtained by essentially four modifications:

1. Data feedback of normalized data from the multiplier M into the aligners AL1 and AL2 via feedback path ND;
2. Shift left one digit by SL1 and SL2 on both sides of the data path for taking account of a possible leading zero digit of the product (special zeroing of guard digits);
3. Exponent generation by 9 bits for overflow and underflow recognition in Z1 and Z2. Due to underflow the exponent result is reset to zero on the fly by true zero; and
4. Both aligners AL1 and AL2 are expanded to 16 digits.

For performing the fast multiply-add instruction sequence the following procedural steps are necessary (please refer to Fig.5):

1. Read the operands OPDI and OPDII for performing a multiplication;
2. Calculate the intermediate exponent product and build the partial sums for multiplication in the multiply array M. At the same time read the operand OPD1 for the addition;
3. Add the partial sums of the multiply array to build the end product and feed the data back for the addition. In parallel a comparison of exponents is performed for an alignment in a 16-digit frame. An end alignment is then adjusted by one left shift if the leading digit of the product is zero. However, the following cases have to be envisaged:
   a) The product is true zero, so the operand coming from the multiplier array M is forced to zero;
   b) the intermediate product exponent is smaller than the OPD1 exponent, then the product is aligned and no further special actions have to be taken; and
   c) the intermediate exponent is greater than the OPD1 exponent, then the addend has to be aligned;

   If the product does not have a leading zero, then the guard digit of the product has to be set to zero. But, if the product has a leading zero, then both operands (the result operand from the multiplication and OPD1) are shifted left by one digit and the 16th digit (in the example of the data flow of Fig.5) of the aligner becomes the guard digit of the result.
4. When both operands are properly aligned, they will be added to the final result of the multiply-add instruction sequence without any need for a wait cycle (as can be seen from a comparison of Fig.3 and Fig.5).

The examples EX.1 - EX.10 (Fig. 6 - 15) described below show for addition, multiplication and multiplication with immediately following addition, where one operand is the result of the preceding multiplication, various conditions under which the results have to be calculated and how the floating-point processor's data flow handles these situations in the different pipeline stages in accordance with Figs. 2a,b, 3 and 5.

In a following first group of examples EX.1 - EX.6 (Fig. 6 - 11) various conditions are shown which may occur during conventional add and multiply operations in a new floating point processor's data flow in accordance with Fig.4.

EX.1 (Fig. 6)

The operands OPD1 and OPD2 (augment and addend) are transferred to the intermediate adder input registers FA and FB during 'operand read'. The operands consist of a fraction value and an exponent. As the exponents 05 and 07 do not match, a right shift of the lower exponent by two positions is necessary for operand alignment. This is done during 'operand alignment'. The underflow value 7 of operand OPD1 is caught by a guard digit GD for being used later when the result (intermediate or final) of the addition has to be build during 'addition'. After alignment by the aligners AL1 and AL2 and after having passed through the shifters SL1 and SL2, and the true/complement unit T/C, which is interconnected between SL1 and zero detector Z1 the operands are stored in the input registers FC and FD of the adder ADD-A. ADD-A generates the intermediate result IR1 shown in example EX.1.

A normalization of the fraction part of the result has to be performed by normalizer NORM-A which results in a truncated normalized fraction, and the exponents are adjusted. The final result/sum is then stored in output register FE. All these above operations are done in the 'addition'-pipeline stage of the floating-point processor.

EX.2 (Fig. 7)

In example EX.2 a further conventional addition is shown where operand OPD2 is smaller than operand OPD1. Therefore, the fraction of operand OPD1 has to be shifted to the right by the difference (4) of the exponents (05, 01) for operand alignment. Again, the guard digit catches the underflow (1) of the shift operation. As the intermediate result IR1 of the addition has three leading zeros, a left shift by 3 is necessary, resulting in an exponent 02 of the final result FR, stored in adder output register FE.

EX.3 (Fig. 8)

In example EX 3, a multiplication is depicted in which the operands OPDI and OPDII have been read into the multiplier input registers MA and MB. As in the previous examples the operands consist of a fraction and an exponent. The partial sums are build in the multiplier array M and intermediately stored in the multiplier output registers PC and PS. In the example the actual values are omitted for convenience reasons. The partial product addition leads to an intermediate result IR2 in which, however, one fraction part has a leading zero. This causes a left shift by 1 and an exponent adjustment 05 -> 04. So, output register FE now contains the truncated, normalized fraction as well as the adjusted exponent.

EX.4 (Fig. 9)

In this example shift operations do not seem necessary after product addition. Only a truncation is required to normalize the number of positions of the final result in output register FE.

EX.5 (Fig. 10)

In example EX.5, operands are shown having negative exponents (-49, -50) and fractions of OPDI larger than of OPDII. The fraction values do not seem to result in an overflow. As the example shows, a shift operation by 1 left of the intermediate result IR2 is only necessary for an exponent adjustment for the final result FR. However, an exponent underflow took place so that FR is true zero.

EX.6 (Fig. 11)

EX.6 shows a very simple example with negative operands where only a truncation of IR2 is necessary for forming the final result FR in register FE.

The following second group of examples EX.7 - EX.10 (Fig. 12 - 15) show the zero wait processing of the multiply-add instruction and the various procedural steps being performed in the different pipeline stages Stage 1 - Stage 4.

EX.7 (Fig. 12)

As can be seen, the multiplication requires three phases A1 - A3, the same number of phases B1 - B3 which are necessary for an addition. The whole operations therefore are performed within four pipeline stages Stage 1 - Stage 4.

During phase A1 both operands OPDI and OPDII are read into the input registers MA and MB of the multiplier array M (pipeline stage 1).

In the next pipeline stage 2, phase A2 the partial sums of the multiplication are build and transferred subsequently into the multiplier output registers PC and PS for being added later. In the same pipeline stage 2, but phase B1 operand OPD1 is read into an intermediate input register FA for adder ADD-A. The old contents of the other intermediate adder input register FB which was left there from a previous normal add instruction is in this case of no interest because the second operand (OPD2) for addition of a multiply-add instruction is being generated in the next pipeline stage 3, phase A3 by adding up the partial sums in adder ADD-M, thus giving the intermediate result IR1 which is fed-back via the feed-back path ND previously explained in context with Fig. 4, forming now operand OPD2.

As is shown in Fig. 4 the operands on their way to adder input registers FC and FD have, if necessary, to undergo alignment operations in aligners AL1, AL2 and shifters SL1, SL2, when the exponents do not match or zeroing operations in T/C, Z1, Z2, when leading zeros, guard digits GD included, have to be removed before the actual addition in adder ADD-A.

Some special situations are shown in EX.7. As shown under ①, IR1 is fed-back via path ND with one extra digit GD (4 bits) in pipeline stage 3. The guard digit (GD = 8) resulted from the product addition which took place in stage 3, phase A3.

Under ② it is shown that the alignment of operand OPD1 caused by a right shift by 3 positions resulted in an extended data width by two GDs (1,1).

At the position of reference mark ③ it is shown that the operand transferred from IR1 to register FD has a leading zero which has to be removed by a left shift so that the resulting exponent Exp is changed from 05 > 04.

During stage 3, phase A3 further the contents of FE is truncated and normalized. This causes an exponent adjustment of -1 (05 > 04). For forming the final result in stage 4, phase B3, therefore a further left shift is necessary for exponent adjustment prior to the final addition operation. The result of the addition, intermediately stored in IR2, however, has still to be truncated and normalized. During this procedure a guard digit, if present has to be removed and the final result has to be transferred to FE, the output register containing the final result FR.

EX.8 (Fig. 13)

In example EX.8 special situations caused by operand values different from those discussed in EX.7 are marked ④ and ⑤.

In ④ an independent zeroing of the guard digit GD in FD is required which is done in stage 3, phase B3.

In ⑤ it is necessary to truncate the fraction part of the operand in FC. This means that no left shift by one digit has to be made, so that only the n + 1 first digits come into addition.

EX.9 (Fig. 14)

In example EX.9 there is a special situation marked ⑥ shown where in stage 3, phase B2 the operand intermediately stored in FD -the result of the multiplication- requires an additional guard digit GD. As the exponents of both operands are already adjusted (both are 05), there is no subsequent shift operation required.

EX.10 (Fig. 15)

In the final example EX.10 it is shown under mark ⑦ how an exponent underflow is handled. An exponent underflow requires one bit (q) more and a cancellation of data feedback via path ND if a true zero situation was detected by the T/C unit.

Claims

1. Arrangement in a floating point processor comprising
- a multiply section (MS) having a first input register (MA) and a second input register (MB) for intermediately storing the operands (OPDI, OPDII) prior to a multiplication in a multiplier (M) the output of which is connected to adder output registers (PC, PS) for intermediately storing the partial sums of the multiplication prior to their addition in a first adder (ADD-M), and a first normalizer (NORM-M) connected to the adder output for normalizing the sum (OPD2) of the partial sums; and
- an add section (AS) having a third (FA) and a fourth input register (FB) for intermediately storing the operands (OPD1, OPD2) for addition, a first (FC) and a second adder input register (FD) for intermediately storing the operands prior to their addition in a second adder (ADD-A), a first aligner (AL1) for operand (OPD1) alignment interconnected between said third input register and a true/complement unit (T/C) for operand true/complement building, which is connected to said first adder input register, a second aligner (AL2) connected to said fourth input register for operand (OPD2) alignment, and a second normalizer (NORM-A) connected to said second adder's output for normalizing the final result,

characterized in, that for performing a fast multiply-add instruction without requiring a wait cycle there is provided:
- a feedback path (ND; Fig.4) connecting the output of said first normalizer to the input of said first and second aligner;
- a first left shifter (SL1) interconnected between said first aligner and said true/complement unit;
- a zero setter (Z1) interconnected between the true/complement unit and said first adder input register; and
- a second left shifter (SL2) interconnected between said second aligner and a second zero setter (Z2) which itself is connected to said second adder input register.

2. Method of performing a fast multiply-add instruction without requiring a wait cycle in an arrangement in a floating point processor in particular in accordance with claim 1, characterized by the following steps:

1. Read the operands (OPDI, OPDII) for multiplication into first (MA) and second input register (MB) of multiplier (M);

2. Build the partial sums by the multiplier (M);

3. Perform the product exponent calculation and reduce the exponent by 1 if the product has a leading zero, and at the same time read operand (OPD1) of the addition;

4. Add the partial sums of the multiplication and feed the resulting intermediate value back to the aligners (AL1, AL2) via feedback path (ND);

5. Compare the exponents of the intermediate value of the product and the addend (OPD1) and perform, if they do not compare a proper alignment;

6. Test whether the following cases do apply:

a. If the product is true zero then the operand feedback from the multiplier is forced to zero;

b. If the intermediate product exponent is smaller than or equal to the addend operand (OPD1) exponent, then the product will be aligned;

c. If the intermediate product exponent is greater than the addend operand exponent, then the addend will be aligned;

7. If the product does not have a leading zero, then a potential guard digit of the

product is set to zero;
8. If the product has a leading zero, then both operands are shifted left by the shifters (SL1, SL2) by 1 digit and the least significant digit of the aligner becomes the guard digit of the result;
9. When both operands are properly aligned, then they will be added by the second Adder (ADD-A) to the final result of the fast multiply-add instruction.

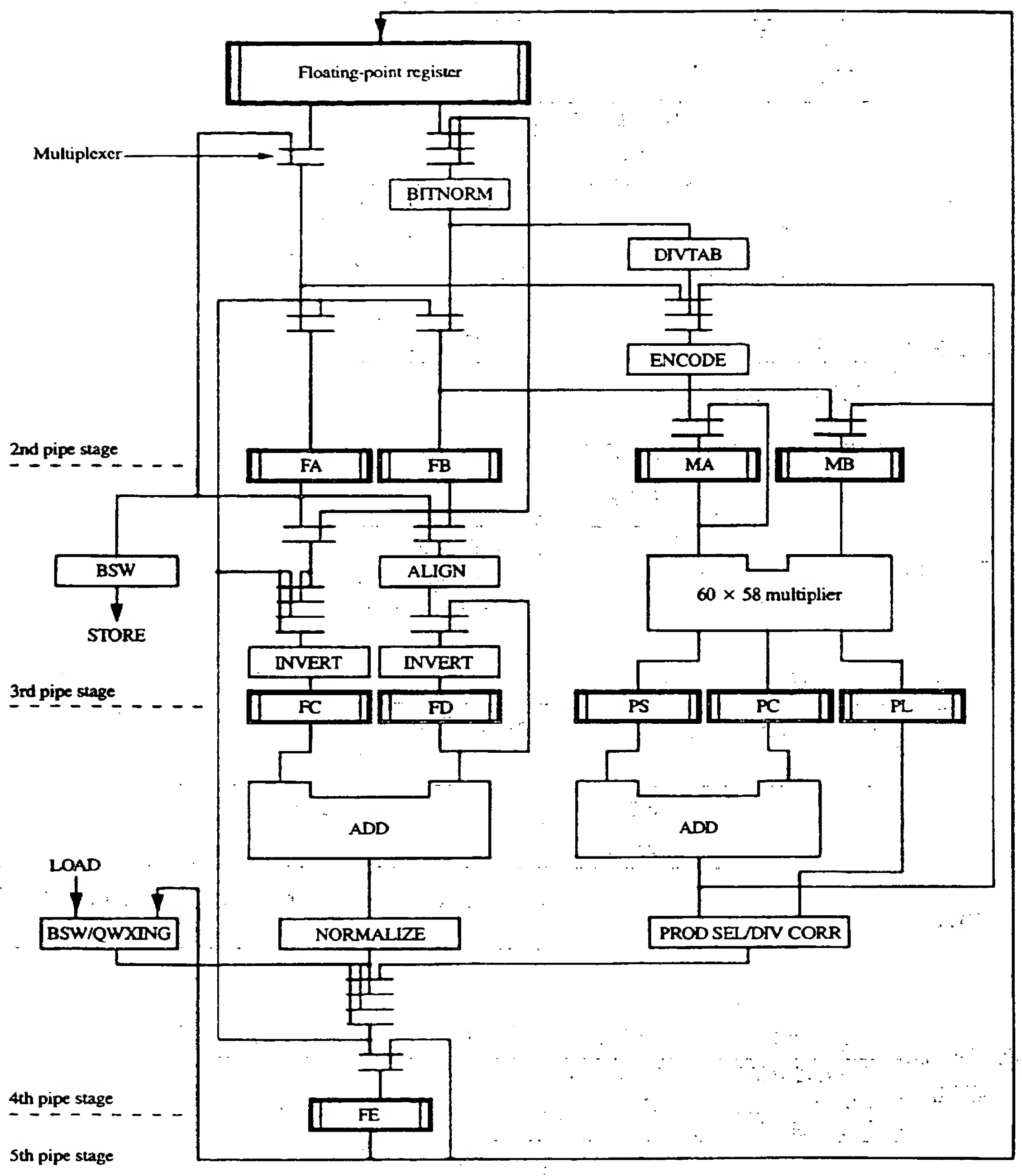

Floating-point register
Multiplexer
BITNORM
DIVTAB
ENCODE
2nd pipe stage
FA
FB
MA
MB
BSW
STORE
ALIGN
60 × 58 multiplier
INVERT
INVERT
3rd pipe stage
FC
FD
PS
PC
PL
ADD
ADD
LOAD
BSW/QWXING
NORMALIZE
PROD SEL/DIV CORR
4th pipe stage
FE
5th pipe stage

FIG. 1

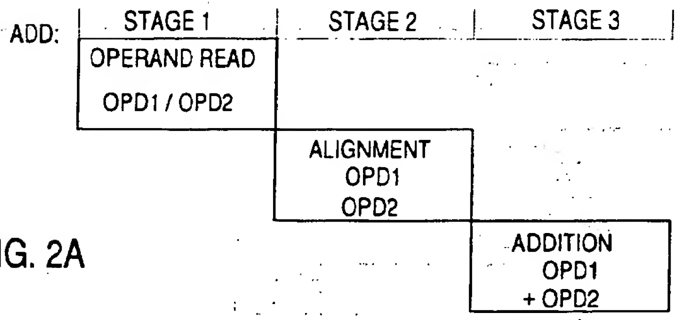


FIG. 2A

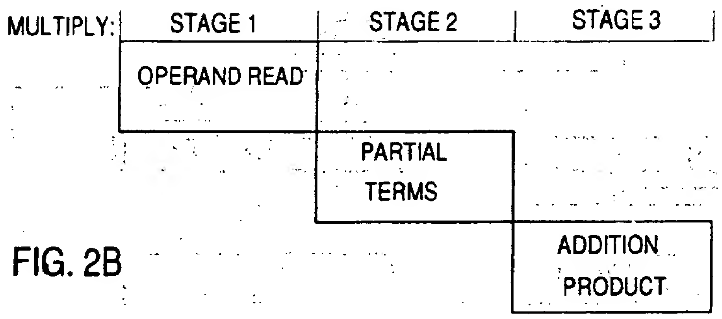


FIG. 2B

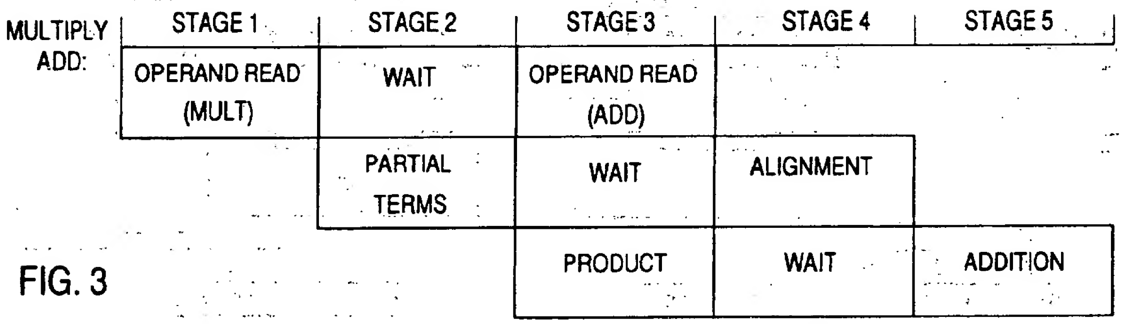


FIG. 3

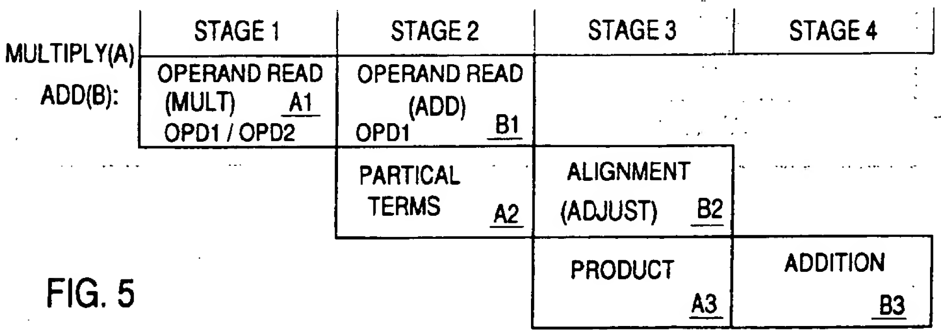


FIG. 5

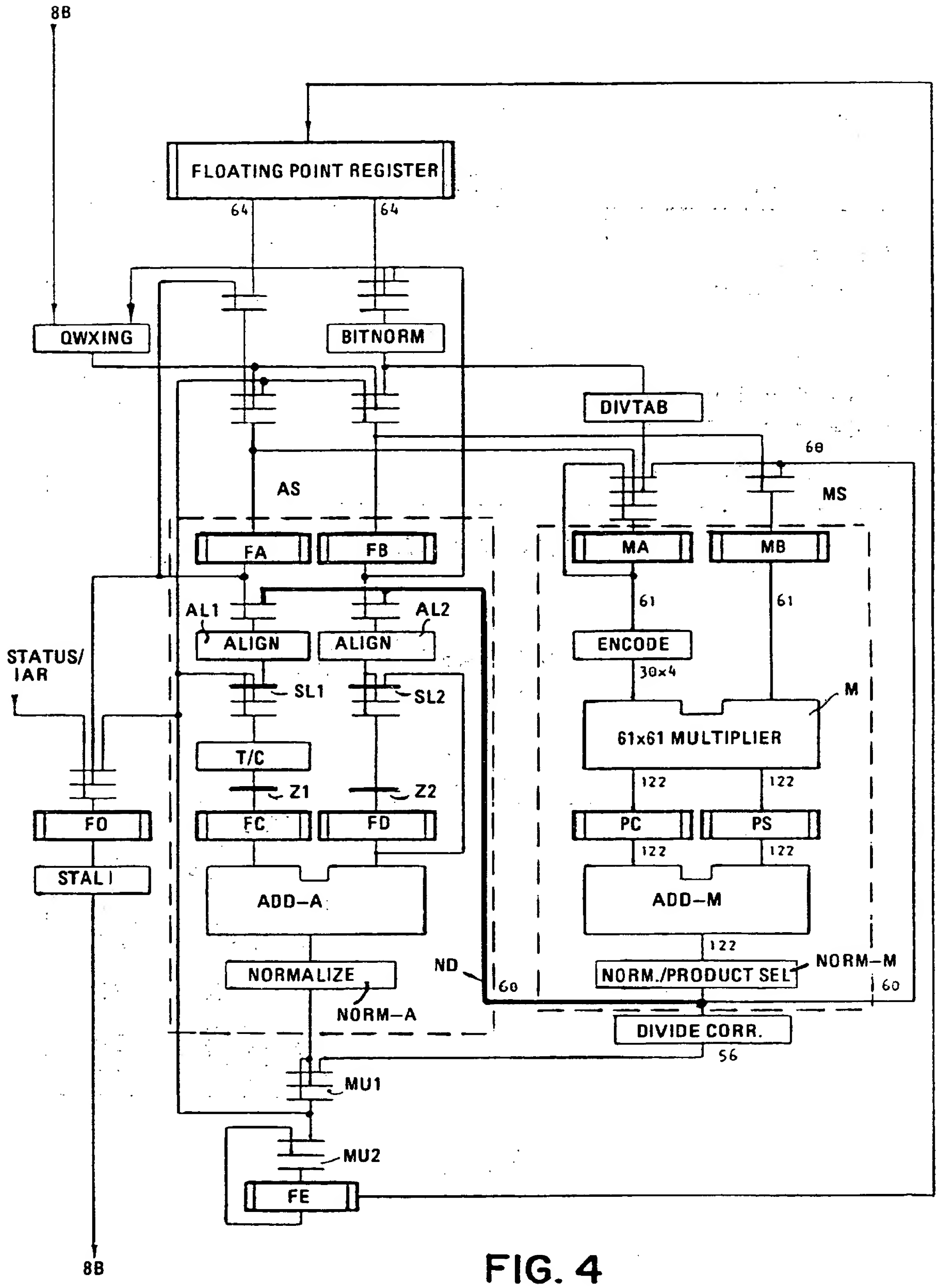

8B
FLOATING POINT REGISTER
64
64
QWXING
BITNORM
DIVTAB
68
AS
MS
FA
FB
MA
MB
AL1
ALIGN
ALIGN
AL2
61
61
ENCODE
30x4
STATUS/IAR
SL1
SL2
M
61x61 MULTIPLIER
T/C
Z1
Z2
122
122
FO
FC
FD
PC
PS
122
122
STALI
ADD-A
ADD-M
122
NORMALIZE
ND
NORM-A
60
NORM./PRODUCT SEL
NORM-M
60
DIVIDE CORR.
56
MU1
MU2
FE
8B

FIG. 4

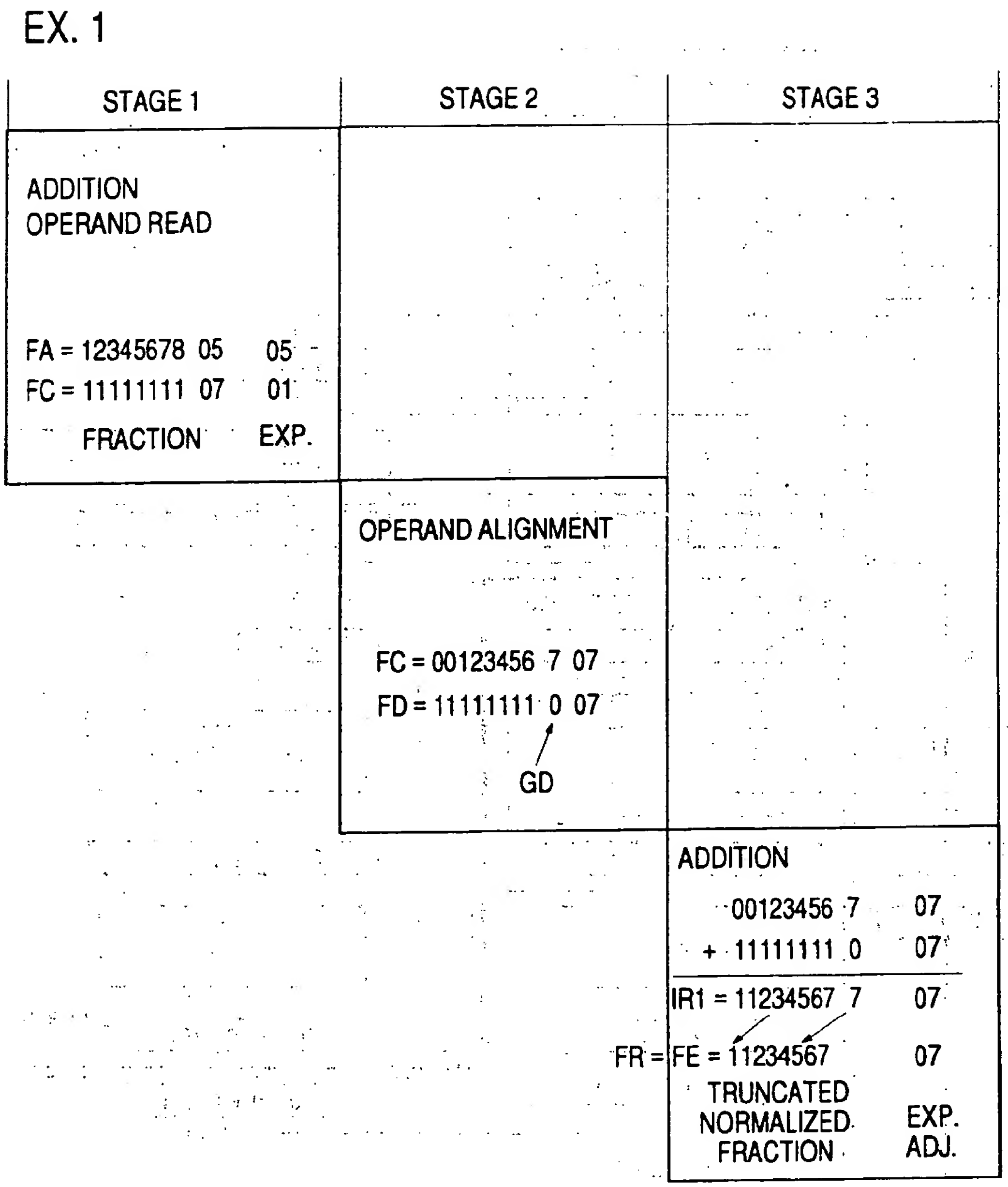

EX. 1
STAGE 1
STAGE 2
STAGE 3
ADDITION
OPERAND READ
FA = 12345678 05 05
FC = 11111111 07 01
FRACTION
EXP.
OPERAND ALIGNMENT
FC = 00123456 7 07
FD = 11111111 0 07
GD
ADDITION
00123456 7 07
+ 11111111 0 07
IR1 = 11234567 7 07
FR = FE = 11234567 07
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.

FIG. 6

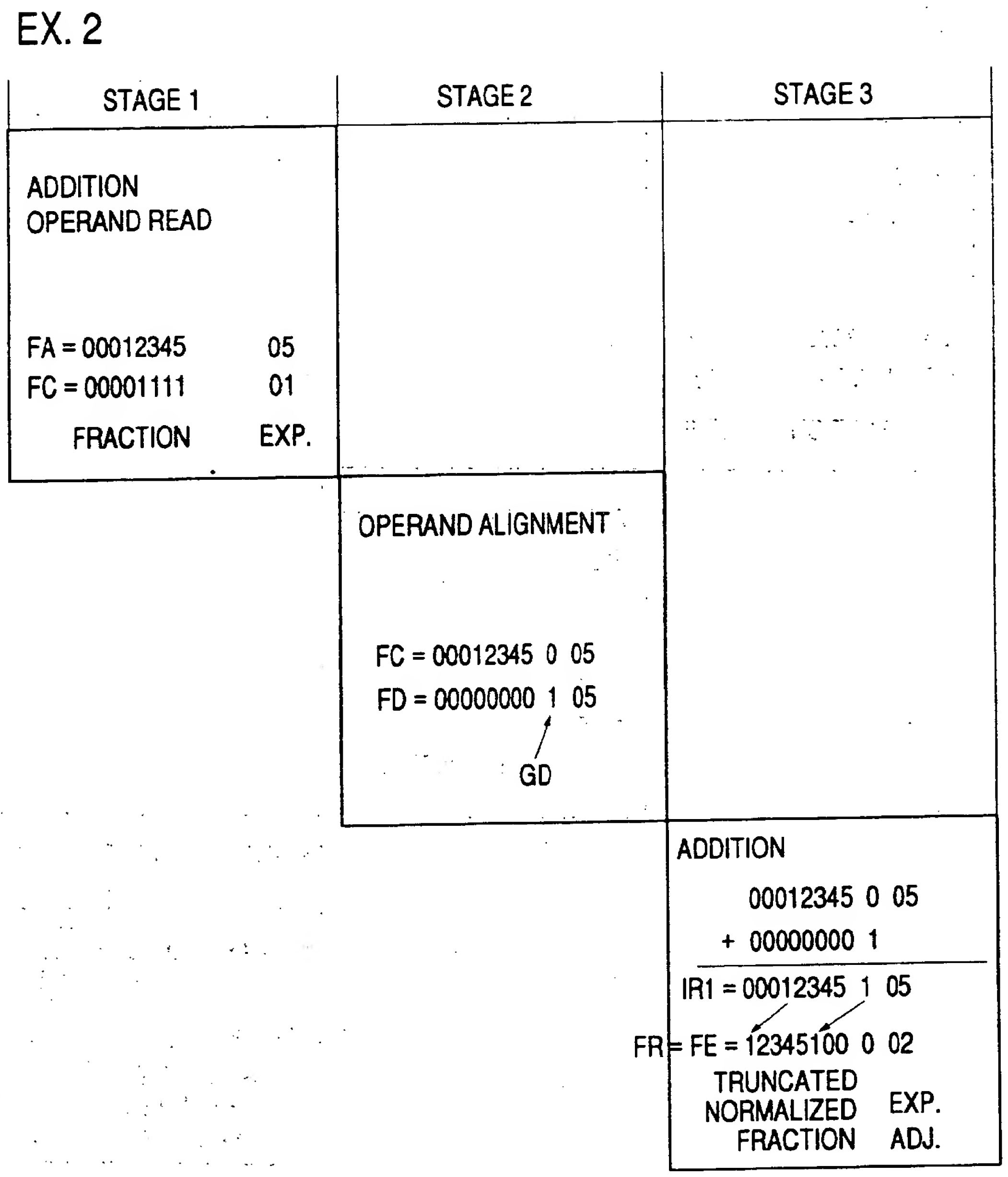

EX. 2
STAGE 1
STAGE 2
STAGE 3
ADDITION
OPERAND READ
FA = 00012345 05
FC = 00001111 01
FRACTION
EXP.
OPERAND ALIGNMENT
FC = 00012345 0 05
FD = 00000000 1 05
GD
ADDITION
00012345 0 05
+ 00000000 1
IR1 = 00012345 1 05
FR = FE = 12345100 0 02
TRUNCATED
NORMALIZED
FRACTION
EXP.
ADJ.

FIG. 7

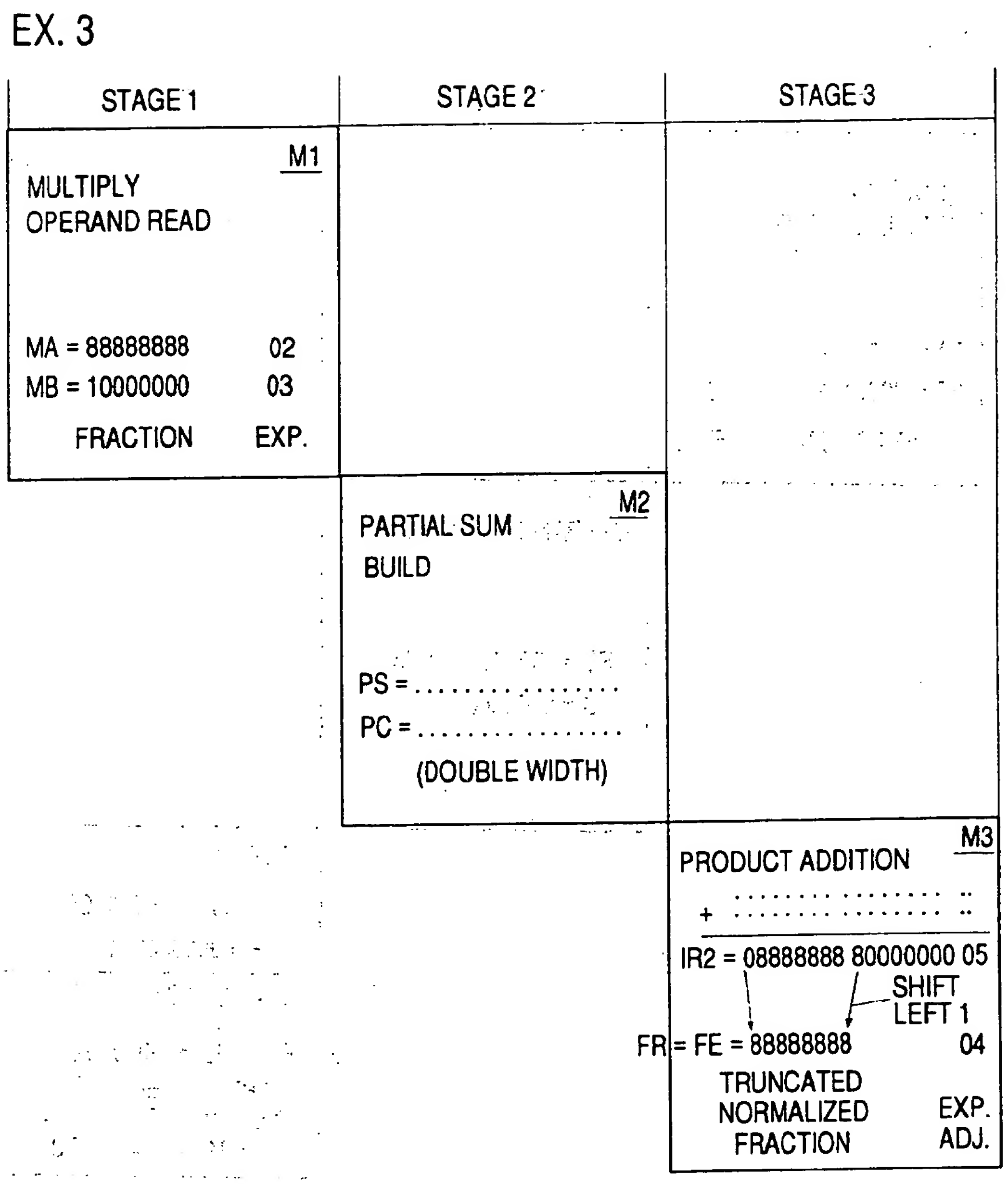

EX. 3
STAGE 1
STAGE 2
STAGE 3
M1
MULTIPLY
OPERAND READ
MA = 88888888 02
MB = 10000000 03
FRACTION
EXP.
M2
PARTIAL SUM
BUILD
PS = ...
PC = ...
(DOUBLE WIDTH)
M3
PRODUCT ADDITION
....
+ ....
IR2 = 08888888 80000000 05
SHIFT
LEFT 1
FR = FE = 88888888 04
TRUNCATED
NORMALIZED
FRACTION
EXP.
ADJ.

FIG. 8

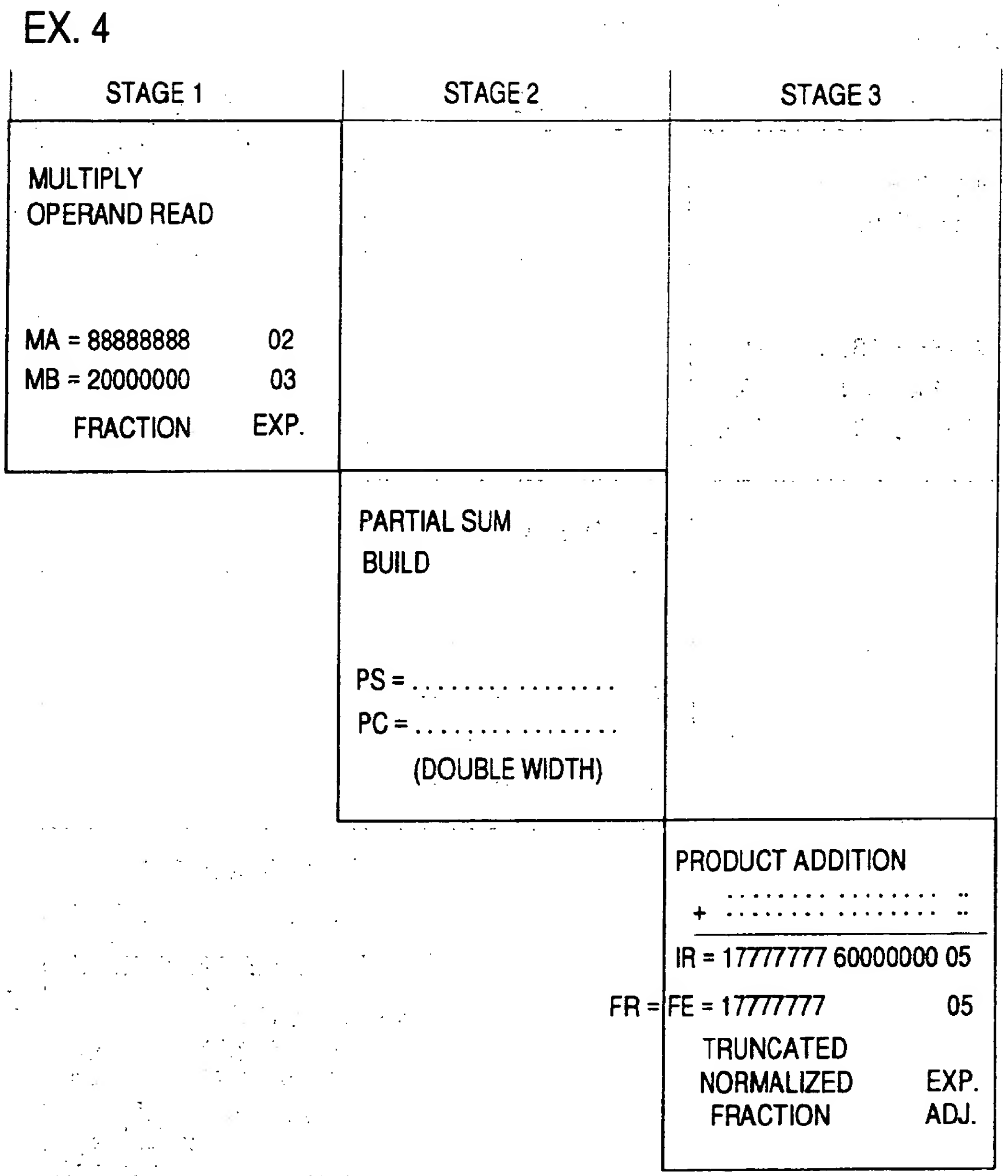
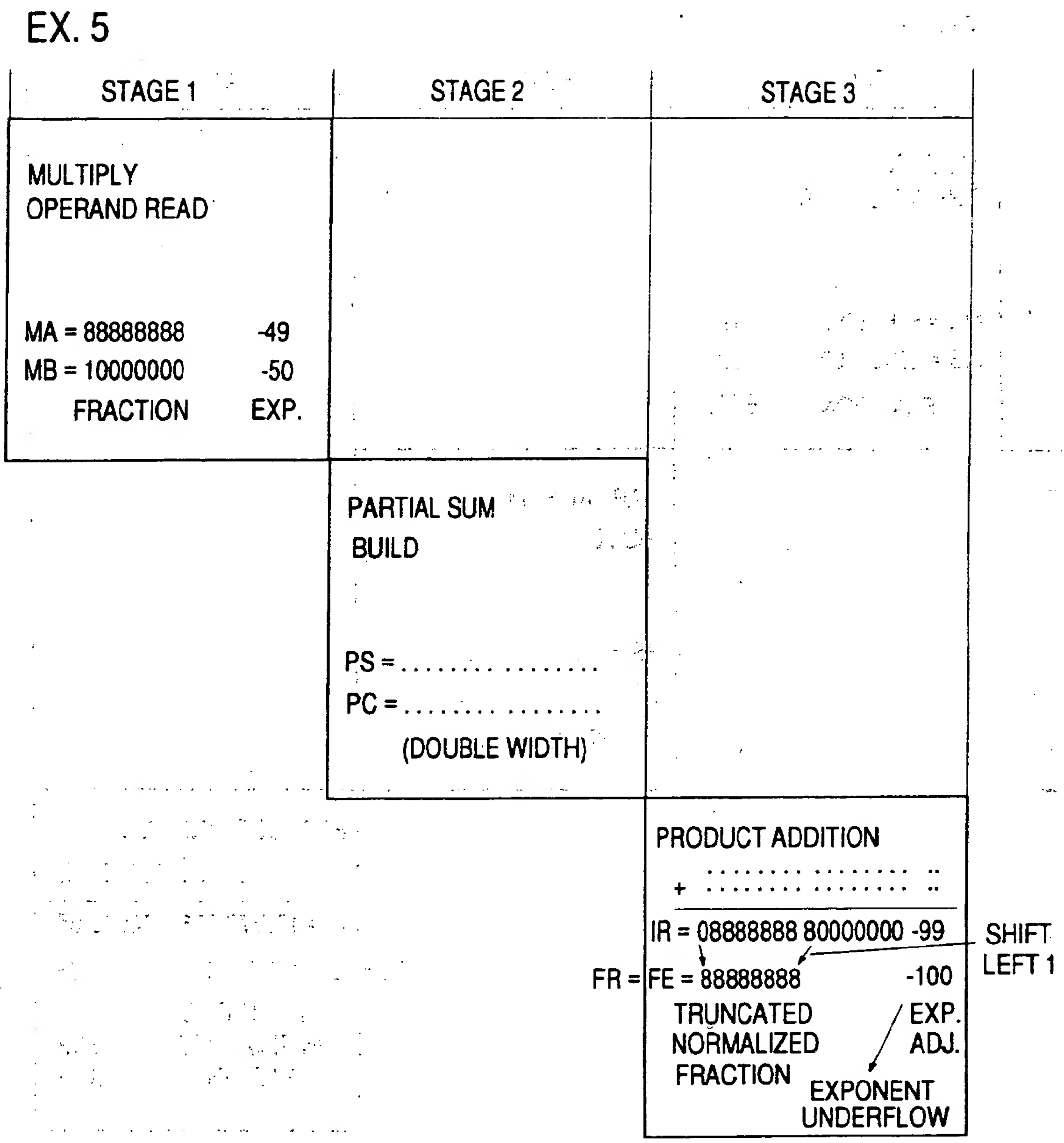

EX. 4
STAGE 1
STAGE 2
STAGE 3
MULTIPLY
OPERAND READ
MA = 88888888 02
MB = 20000000 03
FRACTION
EXP.
PARTIAL SUM
BUILD
PS = ...
PC = ...
(DOUBLE WIDTH)
PRODUCT ADDITION
+
IR = 17777777 60000000 05
FR = FE = 17777777 05
TRUNCATED
NORMALIZED
FRACTION
EXP.
ADJ.

FIG. 9



EX. 5
STAGE 1
STAGE 2
STAGE 3
MULTIPLY
OPERAND READ
MA = 88888888 -49
MB = 10000000 -50
FRACTION EXP.
PARTIAL SUM
BUILD
PS = ...
PC = ...
(DOUBLE WIDTH)
PRODUCT ADDITION
....
+ ....
IR = 08888888 80000000 -99
SHIFT
LEFT 1
FR = FE = 88888888 -100
TRUNCATED
NORMALIZED
FRACTION
EXP.
ADJ.
EXPONENT
UNDERFLOW

FIG. 10

EX. 6

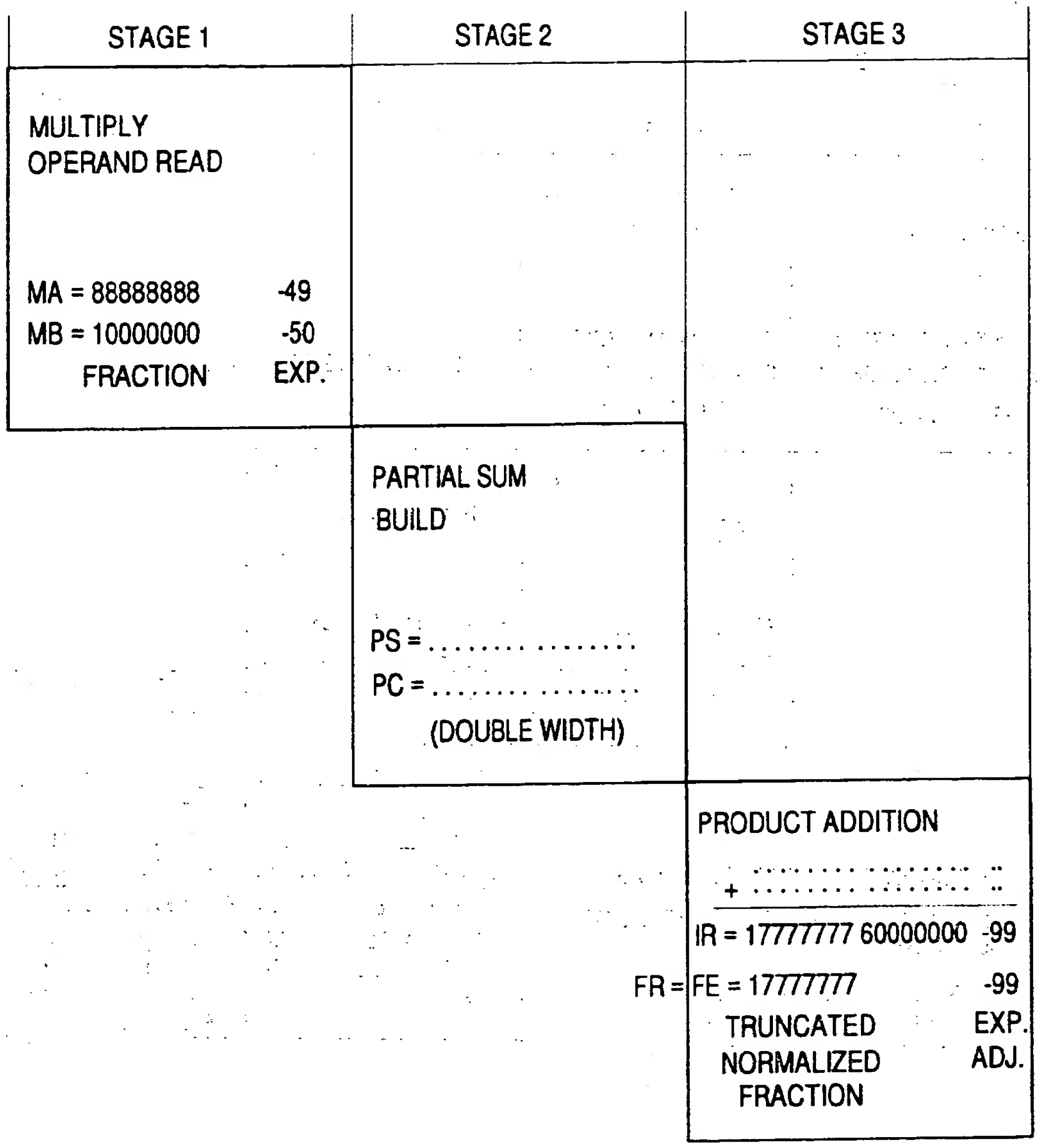

STAGE 1
STAGE 2
STAGE 3
MULTIPLY
OPERAND READ
MA = 88888888 -49
MB = 10000000 -50
FRACTION EXP.
PARTIAL SUM
BUILD
PS = ...
PC = ...
(DOUBLE WIDTH)
PRODUCT ADDITION
+
IR = 17777777 60000000 -99
FR = FE = 17777777 -99
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.

FIG. 11

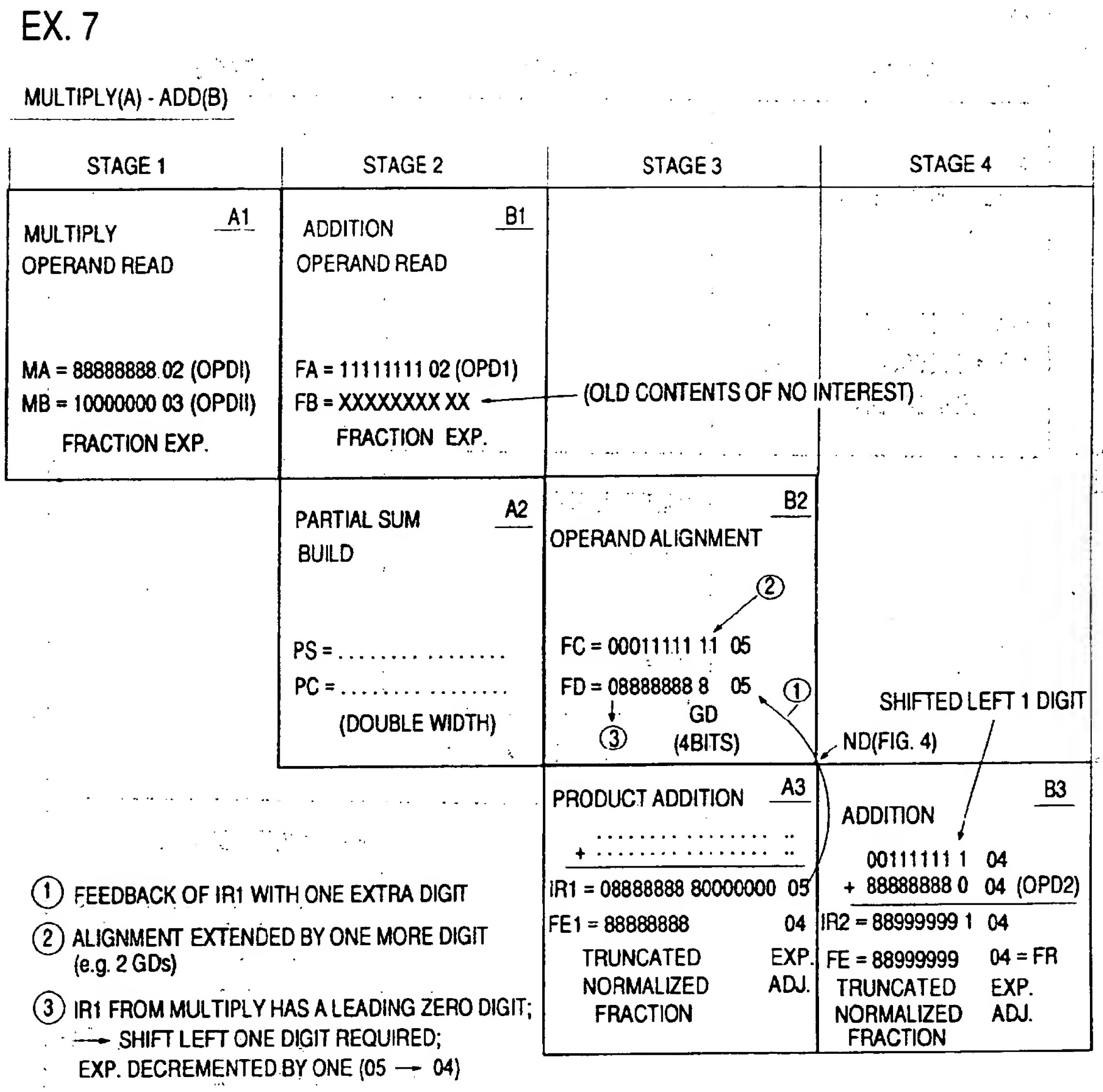

EX. 7
MULTIPLY(A) - ADD(B)
STAGE 1
STAGE 2
STAGE 3
STAGE 4
A1
MULTIPLY OPERAND READ
MA = 88888888 02 (OPDI)
MB = 10000000 03 (OPDII)
FRACTION EXP.
B1
ADDITION OPERAND READ
FA = 11111111 02 (OPD1)
FB = XXXXXXXX XX
FRACTION EXP.
(OLD CONTENTS OF NO INTEREST)
A2
PARTIAL SUM BUILD
PS = ...
PC = ...
(DOUBLE WIDTH)
B2
OPERAND ALIGNMENT
②
FC = 00011111 11 05
FD = 08888888 8 05
①
③
GD (4BITS)
SHIFTED LEFT 1 DIGIT
ND(FIG. 4)
A3
PRODUCT ADDITION
....
+ ....
IR1 = 08888888 80000000 05
FE1 = 88888888 04
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.
B3
ADDITION
00111111 1 04
+ 88888888 0 04 (OPD2)
IR2 = 88999999 1 04
FE = 88999999 04 = FR
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.
① FEEDBACK OF IR1 WITH ONE EXTRA DIGIT
② ALIGNMENT EXTENDED BY ONE MORE DIGIT (e.g. 2 GDs)
③ IR1 FROM MULTIPLY HAS A LEADING ZERO DIGIT; → SHIFT LEFT ONE DIGIT REQUIRED; EXP. DECREMENTED BY ONE (05 → 04)

FIG. 12

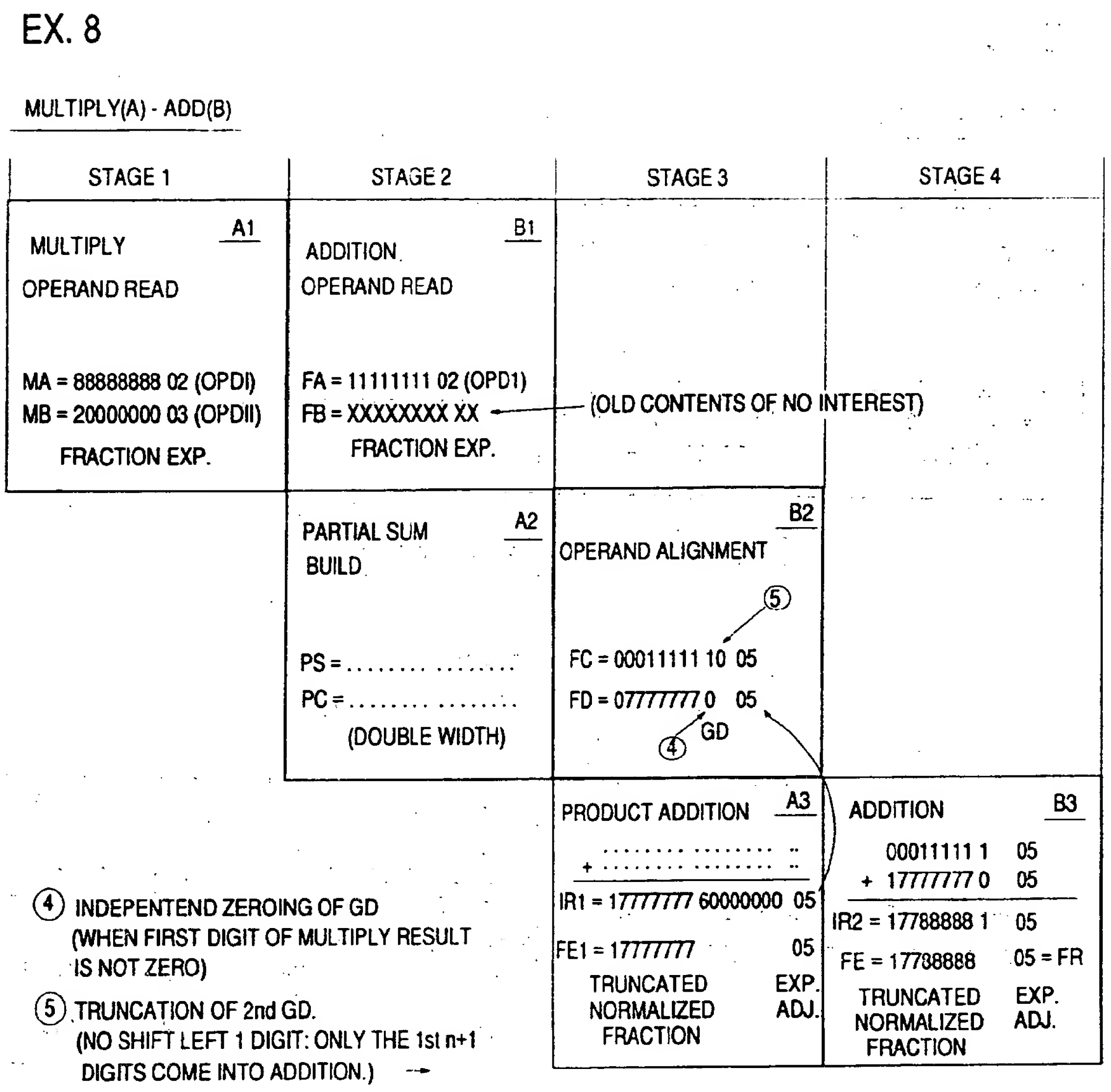

EX. 8
MULTIPLY(A) - ADD(B)
STAGE 1
STAGE 2
STAGE 3
STAGE 4
A1
MULTIPLY
OPERAND READ
MA = 88888888 02 (OPDI)
MB = 20000000 03 (OPDII)
FRACTION EXP.
B1
ADDITION
OPERAND READ
FA = 11111111 02 (OPD1)
FB = XXXXXXXX XX
FRACTION EXP.
(OLD CONTENTS OF NO INTEREST)
A2
PARTIAL SUM BUILD
PS = ...
PC = ...
(DOUBLE WIDTH)
B2
OPERAND ALIGNMENT
5
FC = 00011111 10 05
FD = 07777777 0 05
4
GD
A3
PRODUCT ADDITION
....
+ ....
IR1 = 17777777 60000000 05
FE1 = 17777777 05
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.
B3
ADDITION
00011111 1 05
+ 17777777 0 05
IR2 = 17788888 1 05
FE = 17788888 05 = FR
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.
4 INDEPENTEND ZEROING OF GD (WHEN FIRST DIGIT OF MULTIPLY RESULT IS NOT ZERO)
5 TRUNCATION OF 2nd GD. (NO SHIFT LEFT 1 DIGIT: ONLY THE 1st n+1 DIGITS COME INTO ADDITION.)

FIG. 13

EX. 9

MULTIPLY(A) - ADD(B)

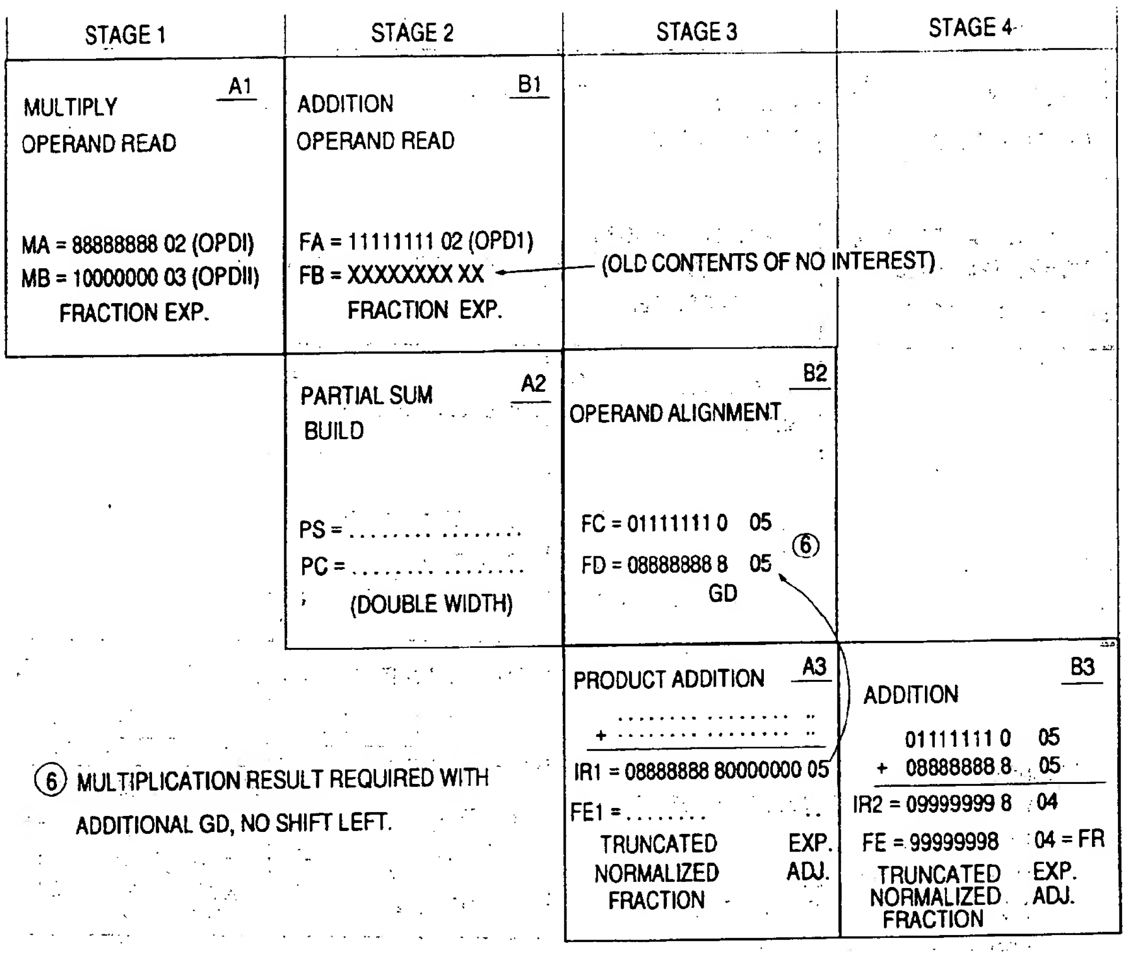


FIG. 14

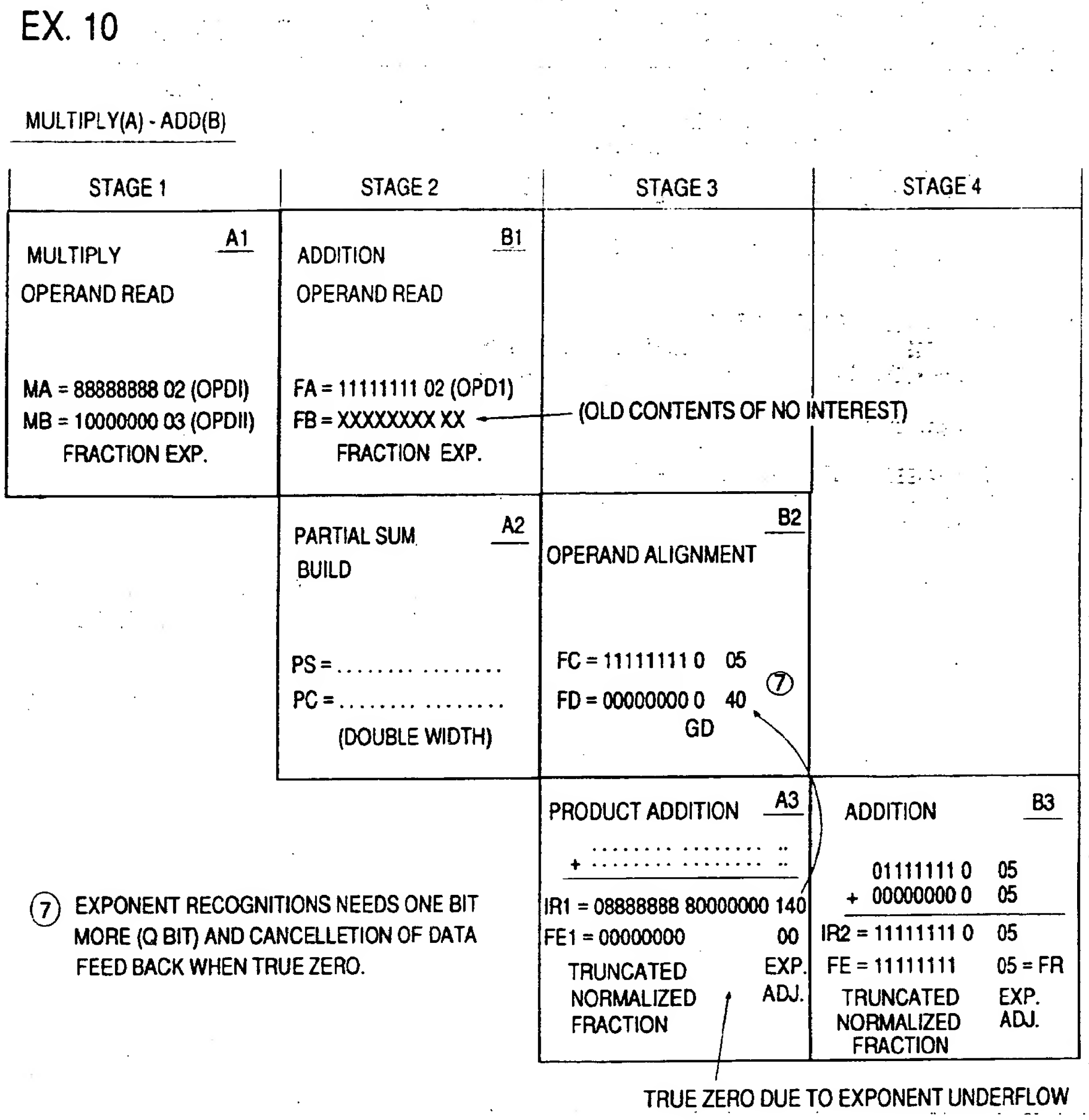

EX. 10
MULTIPLY(A) - ADD(B)
STAGE 1
STAGE 2
STAGE 3
STAGE 4
A1
MULTIPLY
OPERAND READ
MA = 88888888 02 (OPDI)
MB = 10000000 03 (OPDII)
FRACTION EXP.
B1
ADDITION
OPERAND READ
FA = 11111111 02 (OPD1)
FB = XXXXXXXX XX
FRACTION EXP.
(OLD CONTENTS OF NO INTEREST)
A2
PARTIAL SUM BUILD
PS = ...
PC = ...
(DOUBLE WIDTH)
B2
OPERAND ALIGNMENT
FC = 11111111 0 05
FD = 00000000 0 40
GD
⑦
A3
PRODUCT ADDITION
IR1 = 08888888 80000000 140
FE1 = 00000000 00
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.
B3
ADDITION
01111111 0 05
+ 00000000 0 05
IR2 = 11111111 0 05
FE = 11111111 05 = FR
TRUNCATED NORMALIZED FRACTION
EXP. ADJ.
⑦ EXPONENT RECOGNITIONS NEEDS ONE BIT MORE (Q BIT) AND CANCELLETION OF DATA FEED BACK WHEN TRUE ZERO.
TRUE ZERO DUE TO EXPONENT UNDERFLOW

FIG. 15

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 93 11 5706

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
Y	IBM J. RES. DEV. (USA), IBM JOURNAL OF RESEARCH AND DEVELOPMENT, JULY 1992, USA, 36, 4, 733 - 749 ISSN 0018-8646 Dao-Trong S et al 'A single-chip IBM system/390 floating-point processor in CMOS' * page 734, column 1, line 21 - line 43; figure 1 * ---	1,2	G06F7/544
Y	PATENT ABSTRACTS OF JAPAN vol. 013, no. 444 (E-828)5 October 1989 & JP-A-01 170 111 (HITACHI LTD) 5 July 1989 * abstract * ---	1,2	
A	US-A-4 916 651 (GILL ET AL) * column 5, line 27 - line 56; figure 1; table 2 * -----	1,2	
			TECHNICAL FIELDS SEARCHED (Int.Cl.6)
			G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	17 February 1994	Beindorff, W

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)